Using a CSP based programming model for reconfigurable processor arrays

Zain-ul-Abdin and Bertil Svensson

Svensson B, Ul-Abdin Z. Using a CSP based programming model for reconfigurable processor arrays. In: International Conference on Reconfigurable Computing and FPGAs, 2008. ReConFig '08. Los Alamitos, California: IEEE Computer Society; 2008. p. 343-348.

DOI: http://dx.doi.org/10.1109/ReConFig.2008.41
Copyright: IEEE

In Proceedings of International Conference on Reconfigurable Computing and FPGAs (ReConFig08), December 3-5, 2008.

Using a CSP Based Programming Model for Reconfigurable Processor Arrays

Zain-ul-Abdin and Bertil Svensson
Centre for Research on Embedded Systems (CERES),
Halmstad University, Halmstad, Sweden.
{Zain-ul-Abdin,Bertil.Svensson}@hh.se

Abstract

The growing trend towards adoption of flexible and heterogeneous, parallel computing architectures has increased the challenges faced by the programming community. We propose a method to program an emerging class of reconfigurable processor arrays by using the CSP based programming model of occam-pi. The paper describes the extension of an existing compiler platform to target such architectures. To evaluate the performance of the generated code, we present three implementations of the DCT algorithm. It is concluded that CSP appears to be a suitable computation model for programming a wide variety of reconfigurable architectures.

1. Introduction & Motivation

The increased complexity of the next generation media applications, such as voice and video processing as in HDTV, or baseband processing in cell phone base stations, has increased the demands for high performance combined with low power. In order to fulfill the computational demands of these applications, new parallel architectures are emerging. One of the emerging classes of these parallel architectures is reconfigurable processor arrays, which consist of numerous small processors composed in a reconfigurable interconnection network. These arrays achieve performance growth by exploiting parallelism, instead of scaling the clock frequency of a single powerful processor [1].

However, the applicability of reconfigurable processor arrays in embedded and high performance computing will not be useful if programmers do not structure their applications and runtime systems in an efficient way. Furthermore, programs intended for these parallel architectures contain several parts that execute concurrently, which creates dependencies between the processing elements such that one element needs data computed by another element before it can continue with the processing. It is difficult to realize these communication dependencies at compile time, which limits the exploitable parallelism. Another related challenge with the parallel programming is the tools to avoid deadlock, timing and synchronization errors.

A traditional way is to introduce new software tools that can automatically parallelize the existing sequential code and provide some sort of abstraction layer to enhance portability. Threads have been used as an approach to parallel programming, but threads as a model of computation have proved to be highly non-deterministic [2]. A natural way is to use a concurrent programming model that allows the programmer to express computations in a productive manner by matching it to the target hardware, and the language is further supported by a compiler for providing portability across different hardware resources.

Occam is a programming language based on the Communicating Sequential Processes (CSP) [3] concurrent computation model and was developed by Inmos for their microprocessor chip Transputer [4]. occam-pi [5], is a concurrent programming language that combines the ideas of occam with pi-calculus [6]. A program in occam-pi is a composition of processes, where communication between the processes is managed by unbuffered message passing channels.

In earlier work, we have demonstrated the effectiveness of generated HDL code from a CSP based high-level language [7]. In this paper, we present a method for compiling occam-pi programs to reconfigurable processor arrays. The method is based on implementing a compiler backend for generating native code. This work is part of our ongoing research whose initial studies are presented here. The target architecture for initial studies is the Ambric fabric of processors. We present the results of streaming DCT algorithm implementation.

The rest of the paper is organized as follows: Section 2 provides an overview of the Ambric architecture and its programming model. Section 3 provides a review of the occam-pi language. Section 4 describes the compiler and the implementation description, and results are discussed in Section 5. The paper is concluded with some concluding remarks and future work in Section 6.

2. Ambric Architecture and Programming Model

The Ambric fabric is based on the globally asynchronous locally synchronous (GALS) principle and uses the Kahn Process Networks (KPN) [8] model of computation with bounded buffering. The array is composed of two pairs of Compute Unit (CU) and RAM Unit (RU) [9]. The CU consists of two 32-bit Streaming RISC (SR) processors, two 32-bit Streaming RISC processors with DSP extensions (SRD), and a 32-bit channel interconnect for interprocessor and inter CU communications. The RU consists of four banks of RAM along with a dynamic channel interconnect to facilitate communication with these memories.

The architecture was designed to support a structured object programming model as shown in Figure 1, where the individual objects are programmed in a sequential manner in a subset of the java language or in assembly language [10]. Objects communicate with each other using hardware channels without using any shared memory. Each channel is unidirectional, point-to-point, and has a data path width of a single word. The channels are used for both data and control traffic. The primitive objects contain the functionality of the component and can be combined together to form a composite software object. The individual software objects are then linked together using either a visual interface or by using the aStruct language, according to the structure of the application.

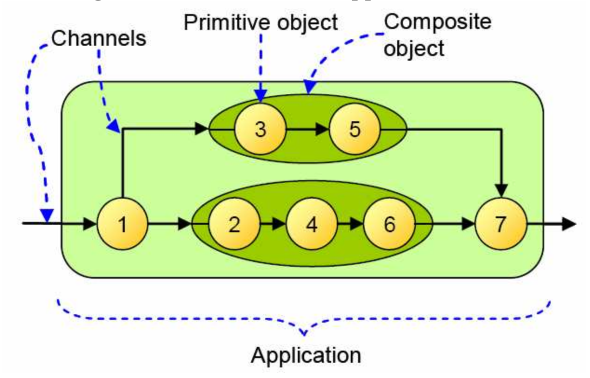

Figure 1. Ambric Structured Object Programming Model (SOPM) [10].

Thus, when designing an application in the Ambric environment, we need to partition the application into a structured graph of objects and define the functions of the individual objects. It is then up to the tools to compile or assemble the source code and to generate the final configuration after completing placement and routing.

3. Overview of Occam-pi

The occam language is based on the CSP [3] process algebra and the main distinguishing feature of occam compared to sequential languages is its ability to compose processes in parallel as easily as in sequential manner. Occam was designed to support distributed computing in the form of application processes that can be executed on either one Transputer or on a number of such devices with very little modifications required. Occam has built in semantics for concurrency and interprocess communication. The channel communications between two processes is a rendezvous, where both processes must cooperate to complete the communication and then continue. The communication between the processes is handled via channels use message passing, which helps in avoiding interference problems.

Occam-pi [5] can be regarded as an extension of classical occam to include the mobility features of the pi-calculus [6]. The mobility feature is provided by the dynamic asynchronous communication capability of the pi-calculus.

3.1. Language primitives

The hierarchical modules in occam are composed of procedures and functions. The granularity of processes varies, and even individual communication and assignment can be regarded as a process. The primitive processes provided by occam include assignment, input process (`?`), output process (`!`), timer process, timeout process, skip process (`SKIP`), stop process (`STOP`), and barrier synchronization process (`SYNC`). In addition to these there are also structural processes such as sequential processes (`SEQ`), parallel processes (`PAR`), alternative processes (`ALT`), `while`, `if/else`, `CASE`, and replicated processes [4].

A process in occam contains both the data and the operations it is required to perform on the data. The data in a process is strictly private and can be observed and modified by the owner process only. This means that the state of the process is only known and controlled by the same process. However, in occam-pi the data can be declared as `MOBILE`, which means that the ownership of the data can be passed between different processes. In classical occam the channel definition is fixed at compile time, but occam-pi offers the possibility of having mobile channels. It is this property of occam-pi that is useful when creating a network of processes that changes its configuration at run-time.

4. Compilation Methodology

In order to compile occam-pi to different computer architectures, we can either use Transterpreter [11], which is a portable and extensible occam-pi run-time system for concurrent language research developed at the University of Kent or generate the native code for the target architecture.

Although the use of Transterpreter as a runtime system for occam-pi programs provides a portable platform, its associated memory requirements and overhead of running a bytecode interpreter does not provide a good solution for the Ambric architecture. To address these issues we propose to generate the native code and thus achieve an efficient implementation while using the safe concurrency properties of occam-pi.

4.1. Tock for Ambric

Translator from Occam to C from Kent (Tock) is a compiler for occam developed in the Haskell language at University of Kent [12]. The implementation of the compiler is based on a nanopass architecture, where the overall structure of the compiler is divided into successive fine-grained passes and each pass performs only a single task. This kind of structure makes the organization of the compiler more logical and simplifies development, testing and debugging. Different compilers have a tendency to detect a wide variety of compile-time errors. For instance, an assembly language compiler can locate syntax errors, as long as the operation is valid, and compilers for high-level languages like C and java can detect type errors and errors caused by potential use of variables before initialization. Similarly, the Tock compiler performs compile-time verification checks, to prevent unsafe parallel usage and look for undefined objects. However, the checks for over/underflow in arithmetic operations, errors related to data type conversions, and array bound checks are performed at runtime. Another distinguishing feature of the Tock compiler is that it is based on test-driven development, which means that for each pass a separate test is also developed to check its correctness.

The construction of Tock is heavily based on the use of monads. In general, the structure of Tock can be divided into three main phases as shown in Figure 2. Each of these phases transforms the output of the previous step into a simpler form while preserving the semantics of the original program. For achieving portability, the compiler is divided into front end, which consists of phases up to machine independent optimization, and back end, which includes the remaining phases that are dependent upon the target machine architecture.

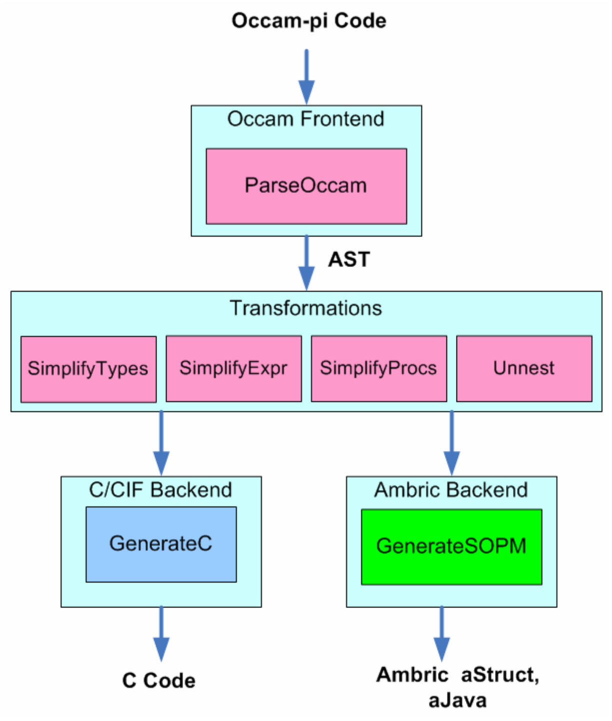

Figure 2. Tock compiler block diagram.

In the following we give a brief description of each of these three phases, including the modifications implemented for supporting the Ambric backend.

4.2. Frontend

The frontend of the compiler, which analyzes the source code in occam, consists of several modules for parsing, syntax and semantic analysis. Steps for resolving names and type checking are also performed at this stage. At first, the lexical analyzer, which is based on Alex [13], scans the source code and converts the strings to tokens. It should also keep track of the indentation during the tokenization process, in which the indentation is converted into markers. Parsing is then performed by using language grammar rules to create Abstract Syntax Trees (AST) from tokens generated at the lexical analysis stage. The parser stage uses a combinator-based parsing library called Parsec [14] that greatly simplifies writing monadic programs. The monadic combinators allow parsing languages described by context-sensitive grammars. Finally, the semantic analysis and type checking is performed to check and resolve any grammar ambiguities.

In the Ambric backend, channels can only be used for accessing channel ends, and channels ends can be used for reading and writing depending on their direction. In order to support the channel end definition required by the Ambric backend we have used the semantics specified by the occam-pi. We have extended the definition of channel type to include the direction whenever a channel name is found followed by a direction token, i.e., "?" for input and "!" for output. In order to implement the channel end definition for a procedure call, we have used the `DirectedVariable` constructor to be passed to the AST whenever a channel end definition is found in the procedure call.

4.3. Transformations

The transformation stage consists of a number of passes either to simplify its input to reduce complexity in the AST for subsequent phases or to convert the input program to a form which is accepted by the backend or to implement different optimizations required by some specific backend. As mentioned, Tock relies heavily on the use of monad transformers. Therefore we describe below some of the most important monad transformers that are used for the Ambric specific transformations.

CompState: The compiler state is represented by a data-type named CompState and there are two monads associated with CompState. The state monad named CSM is used generating unique names for channels and any generated intermediate variable. In order to make only read-only access to the state, we can use the CSMR monad.

PassM: The PassM monad is a stack of four monads; an error monad, a state monad, a writer monad and the IO monad. It is used to transform the function definition in occam-pi to a method in aJava and to avoid wrap-up of PARs to PROCs during the transformation phase. The error monad enables handling of exception-like mechanisms which includes parser errors, type errors and parallel safety problems. The writer monad is used to keep track of warnings during compilation and the IO monad allows the feature of reading in files.

In addition, this phase uses the `Data.Generics` module of GHC to implement a technique to easily query or modify specifically-typed parts of a big data structure without writing tree traversal code manually.

4.4. Ambric backend

The Tock compiler with an Ambric backend added can be regarded as a source-to-source compiler that can produce code from the AST using either the C backend or the Ambric backend. We shall mainly consider here the Ambric backend, but it is worth mentioning here that, in order to reduce the overall code size of the compiler, the Ambric backend still shares some of the passes with the C backend. The Ambric backend is further divided into two main passes.

The first pass generates declarations of aStruct code including the top-level design, the interface and binding declarations for each of the composite as well as primitive objects corresponding to the different processes specified in the occam-pi source code. Before generating the aStruct code, the backend traverses the AST to collect a list of all the parameters passed in procedure calls specified for processes to be executed in parallel. This list of parameters, along with the list of names of procedures called is used to generate the structural interface and binding code for each of the parallel objects.

The next pass makes use of the structured composition of the occam constructs, such as `SEQ`, `PAR`, `ALT`, and `CASE`, which allows intermingling processes and declarations and replication of the constructs like (`SEQ`, `PAR`, `ALT`, `IF`). The backend uses the `genStructured` function from the `generateC` module of the C backend to generate the aJava class code corresponding to processes which do not have PAR construction.

5. Implementation Results & Discussion

In this section, we present and discuss three different implementations of the One-Dimensional Discrete Cosine Transform (1D-DCT), which are developed in occam-pi and then ported to Ambric using our compilation platform.

DCT is a lossy compression technique used in video compression encoders to transform an $N\times N$ image block from the spatial domain to the DCT domain by decomposing the image into spatial frequency components called the DCT coefficients [15]. The mathematics for the 1D-DCT algorithm is described by the following equation:

$$X(k) = C(k)\sum_{n=0}^{N-1} x(n)\cos\frac{(2n+1)k\pi}{2N}$$
$$0 \le k \le N-1$$

where

$$C(0)=\sqrt{\frac{1}{N}} \text{ and } C(k)=\sqrt{\frac{2}{N}} \qquad 1 \le k \le N-1$$

A forward DCT is expressed as a matrix multiplication operation between an $N \times N$ input matrix and the cosine coefficients matrix [15]. We have used a streaming approach to implement the 1D-DCT algorithm, and the dataflow diagram of an 8-point 1D-DCT algorithm is shown in Figure 3. In order to compute the forward DCT, a 12-bit signed vector is applied to the input on the left, and the forward DCT vector is received at the output on the right. The implementations are based on a set of filters which operate in four stages. Following is a brief description of the three different implementations of the DCT algorithm, all based on the dataflow diagram.

Serialized: In the serialized version, all the four stages are implemented as one SEQ block so that they can be translated to be executed on a single processor in Ambric. The results obtained from such an implementation are used for relating it with the results from the parallelized version.

Coarse-grained Parallelized: The coarse-grained parallelized version implements the computational tasks into four stages in a pipelined manner. Each stage takes eight values from its previous stage, performs the computation and supplies the result of eight values to the next stage.

Fine-grained Parallelized: The fine-grained parallelized version is a fully parallelized implementation of the pipelined version. The computations in each stage of the dataflow diagram are implemented in a PAR block so that they can be executed concurrently on individual processors in Ambric brics. There is also splitter and joiner processes used for channeling data values both in and out of the DCT design.

The cycle count results of the three implementations are summarized in Table 1, where N_P stands for number of processors, C_{LFO} represents latency in cycle count from first input provided to the first output emitted, and the last column shows the throughput of the different implementations relative to the throughput of the serial version indicated by T_S, where T_S is equivalent to producing eight output values every 228 clock cycles.

The staged implementation of the DCT algorithm makes it easier to transform it to a multi-processor design. In the serialized version, all the stages execute on a single processor and thus it takes 228 clock cycles to produce the 8 point DCT results. The coarse-grained parallelized implementation performs the same computation on four processors in a pipelined manner with a fifth memory object generated by the tool to provide buffering before the output.

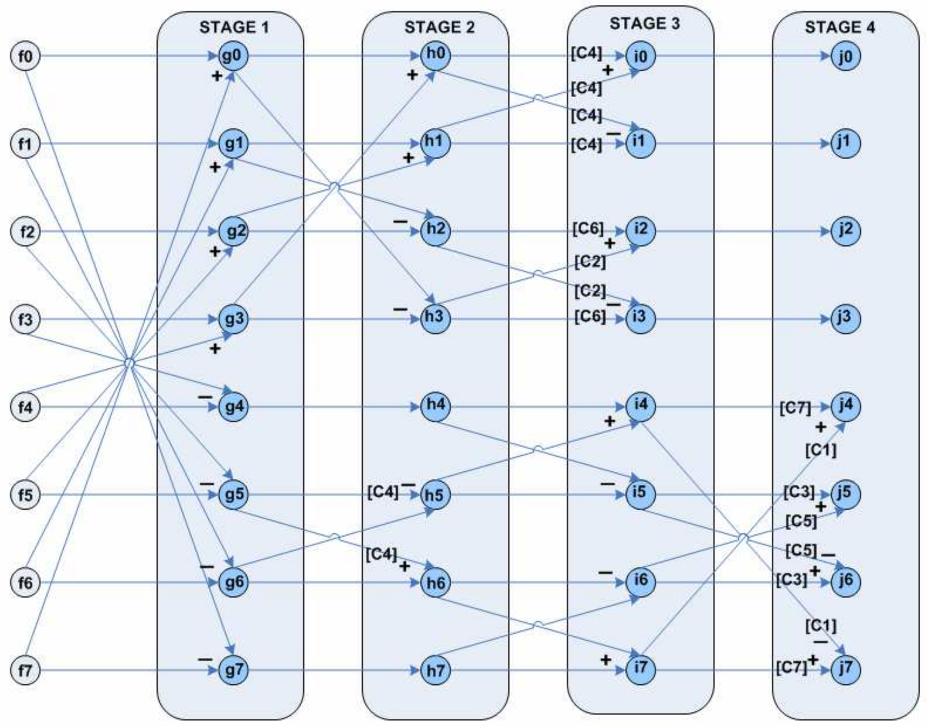

Figure 3. Dataflow diagram for 1D-DCT.

Table 1. Performance results of 8-point DCT implementations.

	N_P	C_{LFO} cycles	Throughput
Serial DCT	1	220	T_S
Coarse-grained Parallelized DCT	4	51	$3T_S$
Fine-grained Parallelized DCT	34	33	$27T_S$

The results depict a latency of 18 clock cycles before the first DCT output is emitted, showing a 4x improvement when compared to the serialized version. Once the pipeline is filled, the implementation provides DCT outputs at three times faster rate. The fine-grained implementation on the other hand is a fully parallelized version that uses almost 8 processors per stage and has a throughput that is 27 times better than the serialized version. The latency of the fine-grained version shows 1.5x improvement when compared to the coarse-grained version and 7x improvement when compared to the serialized implementation.

In terms of lines of code metric, the occam-pi code is three times shorter than its corresponding code in aStruct and aJava. From the programmability point of view, it is observed that programming in occam-pi exposes the communication requirements in an application in a more profound way compared to programming in a combination of aStruct and aJava. Knowing the communication dependencies, the programmer can parallelize the design according to the available resources and performance requirements.

6. Conclusions & Future Work

We have presented our ideas about using CSP as a computation model for programming the emerging class of coarse-grained reconfigurable computing architectures. The ideas are demonstrated by a working compiler, which compiles occam-pi programs to an array of processors, Ambric. An application study is also performed and the results show three different ways to implement the 1D-DCT algorithm, which are compared on the basis of performance versus resources requirements.

We believe that the compositional nature of the process-oriented programming enhances the programmer's understanding and helps in the interaction among different programmers for developing larger designs. Raising the abstraction level for the programmer, while not compromising the performance benefits, will be the key to success for the adoption of such reconfigurable architectures in the mainstream computing industry.

In the future we plan to extend the compiler platform to use the mobility features of occam-pi for implementing reconfigurable logic and to support heterogeneous reconfigurable processing elements.

References

[1] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, and K.A. Yelick, "The landscape of parallel computing research: A view from Berkeley", Tech. Report: UCB/EECS-2006-183, December 2006.

[2] E.A. Lee, "Problem with threads", Technical Report: UCB/EECS-2006-1, January 2006.

[3] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.

[4] Occam® 2.1 Reference Manual, SGS-Thomson Microelectronics Limited, May 12, 1995.

[5] P.H. Welch, and F.R.M. Barnes, "Communicating mobile processes: introducing Occam-pi", *Lecture Notes in Computer Science*, Springer Verlag, Vol. 3525, April 2005, pp. 175-210.

[6] R. Milner, J. Parrow, and D, Walker, "A calculus of mobile processes", *Journal of Information and Computation*, Vol. 100, 1992, pp.1-77.

[7] Zain-ul-Abdin, and B. Svensson, "A Study of Design Efficiency with a High-Level Language for FPGAs", In *Proceedings of 14'th International Reconfigurable Architectures Workshop (RAW'07),* March 2007.

[8] G. Kahn, "The semantics of a simple language for parallel programming", *Information Processing*, North-Holland Publishing Company, 1974, pp. 471-475.

[9] A.M. Jones, and M. Butts, "TeraOPS hardware: A new massively-parallel MIMD computing fabric IC", In *Proceedings of IEEE Hot Chips Symposium*, August 2006.

[10] M. Butts, A.M. Jones, and P. Wasson, "A structural object programming model, architecture, chip and tools for reconfigurable computing", In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 23rd April 2007.

[11] C.L. Jacobsen, and M.C. Jadud, "The Transterpreter: A Transputer Interpreter", *Communicating Process Architectures 2004*, Concurrent Systems Engineering Series, Vol. 62, IOS Press, September 2004, pp. 99–106.

[12] "Tock: Translator from Occam to C by Kent", 8th July, 2008. https://www.cs.kent.ac.uk/research/groups/sys/wiki/Tock

[13] "Alex: Lexer generator for Haskell", 8th July, 2008. http://pages.cpsc.ucalgary.ca/~gilesb/alex/alex.html

[14] D. Leijen, and E. Meijer, "Parsec: A practical parser library", *Electronic Notes in Theoretical Computer Science*, Vol. 41(1), 2001.

[15] XAPP610, "Video Compression using DCT", Xilinx homepage, 16th September 2006. http://direct.xilinx.com/bvdocs/appnotes/xapp610.pdf